Transformation of Path Information for WCET Analysis during Compilation

Raimund Kirner, Peter Puschner
Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1
A-1040 Wien, Austria
{raimund,peter}@vmars.tuwien.ac.at

Abstract

Performing worst-case execution time (WCET) analysis on machine code with program path annotation provided at high-level source code level requires the transformation of path annotations from the source-code level to assembly/object-code level. This path-information transformation can be done outside or integrated into the compiler during code compilation. The first approach is easier to implement but lacks for the support of strong code optimizations performed by the compiler because the external tool would have to make guesses about optimizations. In this paper we present an approach for the program code compilation that integrates the transformation of program path information into the compiler. Path information is transformed through all compiler stages to the adequate path information for the corresponding assembly code level. The WCET analysis tool processes the program at assembly code level with the correctly transformed program-path information to obtain accurate runtime bounds. Several experiments were performed to demonstrate the importance of supporting the transformation of path-information in aggressively optimizing compilers.

1 Introduction

WCET analysis has to deal with two main subproblems, the problem of determining feasible and infeasible execution paths of an application program and the problem of modeling the worst-case behavior of executable code on a given hardware platform.

Within the last years researchers produced valuable solutions in both mentioned areas. With respect to modeling of hardware, WCET research does not just provide solutions for simple CPU and hardware architectures as documented in [10, 14, 17]. Recent work produced a variety of results for modeling the effects of performance-improving mechanisms of today's processors on WCET. The proposed techniques take into account instruction pipelines [7, 12, 20], instruction and data caches [12, 13, 20], parallel execution of instructions [20], and branch prediction [2].

A number of publications address the automatic computation of loop bounds and information about feasible and infeasible execution paths.Various approaches have been taken in this area, including [8].

All techniques that derive path information automatically have only a limited capability to do so. This is due to the fact that the problem of determining execution paths automatically is in general undecidable. For those cases in which complete information about feasible and infeasible paths cannot be generated automatically, the programmer (or code generation tools, if code is produced automatically) has to provide the missing information [12, 13, 14, 17].

The following representation problem occurs when dealing with programmer-supplied information about execution paths. The programmer communicates with the WCET tools at the source-code level, i.e., he or she describes information about (in)feasible paths in terms of the source program. On the other hand, WCET analysis needs to operate on the machine program as produced by the compilation process. During the compilation the program is usually transformed and optimized. Thus the logical structure of the machine program differs significantly from the control structure of the source program. Due to the structural changes caused by the code compilation, mapping path information from the source-code representation to the machine-level representation tends to be non-trivial.

Since the source codes of industrial-strength compilers are not available to real-time researchers, current solutions to the mapping problem are rather built *around* existing compilers than into them. The following overview shows some solutions of structure mapping during compilation.

Co-transformation [3] is an approach of mapping execution information from the source code of a program to the object code for the purpose of WCET analysis. It analyses the source code to obtain information about possible execution paths and loop bounds. A language called *Optimization Description Language* (ODL) was designed to describe the transformation effects of the compiler. Additional information about the performed transformations of the program structure are provided by the slightly modified compiler. This approach can handle several problems introduced by the use of optimizing compilers. A similar approach for the structure mapping is used in [13].

An example for the integration of the WCET analysis into the compiler is given by the academic Modula/R compiler [21] for the MARS architecture [4, 11, 19]. The programming language Modula/R is Modula with extensions like iteration bounds for loops and marker/scopes [16] for WCET analysis and constructs for tasks and messages to support the MARS environment.

An experiment for the structure mapping between high-level source and assembly source code is described by Exler [5]. Exler tries to map path information without using additional information from the compiler. Therefore the mapping cannot be done for several types of compiler optimizations. The approach models a subset of compiler optimizations to estimate the correct mapping. Exler concludes, however, that it is not possible to find an universal mapping algorithm for mapping the program structure without the support by the compiler.

This paper presents an approach that solves the problem of structure mapping during compilation by integrating the mapping process into the compiler. Therefore an existing C compiler is extended to handle (also) the information about the program timing behaviour. The structure mapping is done completely by the compiler. No assumptions about the compiler's compilation technique have to be made by the user.

The paper is structured as follows: Section 2 gives an overview of the WCET analysis tool chain and the syntax of the programming language that was defined to include timing information. Section 3 presents the integration of the structure mapping into the compiler. Section 4 shows the translation of an example program and the result of the WCET analysis for this program, using our path-transformation technique. Finally, Section 5 presents conclusions and plans for future work.

2 Framework and Assumptions

2.1 General Structure of the WCET tool

The WCET analysis environment consists of several logically separated function blocks. It is the aim of this paper to propose a set of tools rather than a single tool to enable composability and software re-use. Solutions of a monolithic WCET tool as described in [21] can probably lead to simpler implementations since all interfaces can be designed for WCET analysis and integrated into a single tool. But this approach tends to produce much more implementation work since all components have to be implemented for the designed internal data structures.

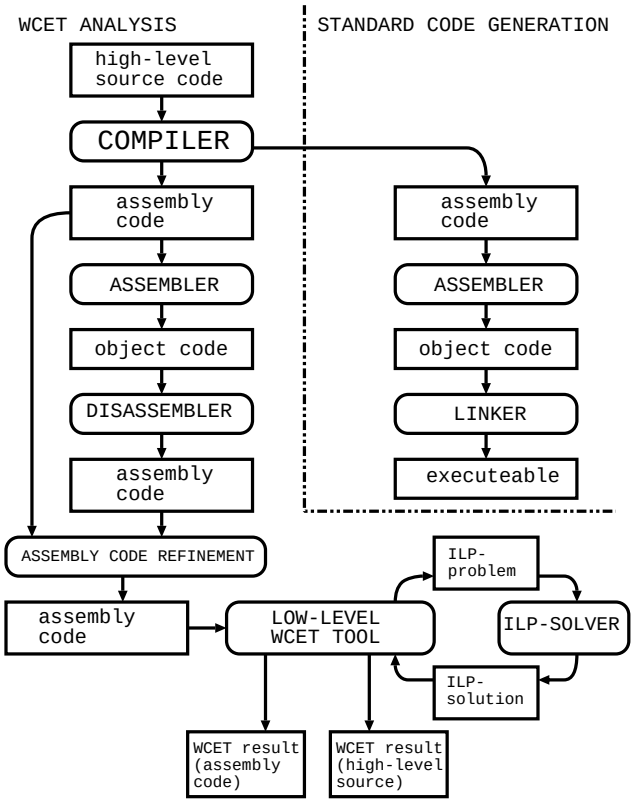

Figure 1. Framework for WCET Analysis

As written above, the approach in this paper uses several tools for specific functions inside the evaluation process. Some of them are standard software tools with slight or no modifications at all.

The whole framework for WCET analysis is shown in Figure 1. Tools that process information or data are displayed as ovals. Rectangles represent the processed information or data. The WCET analysis is shown in the left part of the figure, labeled *WCET analysis*. The right part, called *standard code generation*, is the functionality of a normal program development framework without WCET analysis.

2.1.1 Translation of the Source Code

This part covers the translation of the high-level source code and its WCET information into an annotated assembly source file. The corresponding tools in Figure 1 are the compiler, assembler, disassembler and the instruction matching tool.

The Compiler: The compiler parses a program written in WCETC (see Section 2.2 for details) which is annotated with WCET path-information. The compiler records and matches this information to the corresponding intermediate code of the program. The compiler also keeps track of this relationship through all transformation and optimization steps up to the output of the assembly source.

The Assembler: The assembler produces the object code of an assembly source file. It is used in combination with a disassembler to generate an assembly source without pseudo instructions.

The Linker: The linker is only used for code generation but not for WCET analysis.

Assembly Code Refinement: Some assembly instructions emitted by the compiler are pseudo instructions. They are resolved by the assembler. Such a pseudo instruction can be translated into different machine codes dependent on the whole program. Examples are branch instructions where the substitutes for pseudo instructions depend on how far the branch destination is located from the branch.

2.1.2 WCET Analysis

WCET analysis calculates safe upper bounds for the worst-case execution time of a program. The analysis is done at machine-code level. The WCET results are back-annotated both to the high-level source code and the assembly code.

Low-Level WCET Tool: The low-level WCET tool performs WCET calculations for given assembly source files. The source files have to contain WCET annotations similar to the hardware-independent intermediate code of the compiler as described in Section 3.

This tool parses the assembly source and all comments that contain WCET annotations. With these annotations it constructs a linear programming problem.

ILP Solver: Integer linear programming (ILP) [1] is used to calculate the WCET of a program. The number of restrictions of the ILP problem depend on the number of basic blocks inside the program.

Back-Annotation: The back-annotation transforms the WCET information calculated for a program to a representation which is readable for the user. The result could be for example embedded into the source file as it is shown in Figure 1.

2.2 Programming Language and Notation for Describing Path Information

The presented approach is based on additional information about the runtime behaviour of the program to enable automatic WCET analysis. This additional information is integrated into the programming language which we call WCETC. The following criteria have directed the design of the syntax of WCETC:

- The language should be derived from a commonly used programming language to increase its acceptance and reduce the learning effort for programmers.
- It has to provide annotations to describe the execution paths of the program for static WCET analysis. The minimum annotations would be loop bounds to describe the maximum execution frequencies of loop statements.
- To enhance the quality of the calculated WCET, the language should have additional annotations to describe (in)feasible paths.

To meet the first requirement the syntax of WCETC is derived from ANSI C with several extensions and restrictions (e.g.: no use of `goto` ..., see [9] for details). More precisely it is a superset of a subset of ANSI C. The annotations about the timing behaviour are used to restrict the possible execution paths through the *program flow graph*, a dynamic data structure, which is generated from the analyzed program code by the WCET analysis tool.

```
DO_LOOP: "do"
         "maximum" __int_const__ "iterations"
         C_STMT
         "while" "(" __expression__ ")" ";"
C_STMT: __standard_C_stmt__ | SCOPE
```

Figure 2. Syntax of `do`-Loops in WCETC.

```
SCOPE: "scope" IDENTIFIER "{"
       MC_STMTS
       RESTRICTIONS
       "}"
MC_STMTS: MC_STMT MC_STMTS | $
MC_STMT:  MARKER | C_STMT
MARKER:   "marker" IDENTIFIER ";"
C_STMT:   __standard_C_stmt__ | SCOPE

RESTRICTIONS: RESTRICTION RESTRICTIONS | $
RESTRICTION:  SIMPLE_RESTRICTION | ALT_RESTRICTION

SIMPLE_RESTRICTION: "restriction"     RESTRICTION_EXPR ";"
ALT_RESTRICTION:    "alt_restriction" ALT_EXPR_LIST ";"

ALT_EXPR_LIST: RESTRICTION_EXPR "," ALT_EXPR_LIST |
               RESTRICTION_EXPR
RESTRICTION_EXPR: RESTRICTION_LIST RESTRICTION_LIMIT
RESTRICTION_LIST: MULT_MARKER "+" RESTRICTION_LIST |
                  MULT_MARKER
RESTRICTION_LIMIT: RELOP RESTRICTION_LIST |
                   RELOP INT_CST
MULT_MARKER: "(" INT_CST ")" "*" IDENTIFIER |
             IDENTIFIER
```

Figure 3. Syntax of Markers/Scopes in **wcetC**.

2.2.1 Notation for Loops

For static WCET analysis the information about loop bounds is mandatory. This information cannot be extracted from the source code in general. It is given explicitly in the header of every loop by a constant number. The language WCETC supports three types of loops which are derived from the standard `do`-, `while`- and `for`-loop.

An example of these extensions is given for the `do`-loop in Figure 2.

2.2.2 Notation for Feasible Paths

Information about (in)feasible program paths is used to improve the quality of the calculated WCET of a program. Infeasible paths are paths through the program flow graph that cannot occur during program execution. They are described by the use of markers and scopes [17].

The markers are used to label an edge of the execution path of the program flow graph. These labels are used to express restrictions for the execution count of edges of program-flow paths for each WCET scope (see below). The declared markers are not bound to a certain scope. They can be used for restrictions in any scope enclosing them. The term scope used above is different from the standard scoping scheme of definitions as used in programming languages. Here, a scope defines a code range, on which restrictions can be defined.

The semantic of restrictions is to declare the maximum execution count of certain paths in the flow graph (indicated by a marker) in relation to the execution count of the start node of the scope. Therefore it is, for example, possible to surround two sequential loops with a scope and declare the bound of both loops dependent on each other. The latter concept was also introduced as *"loop-sequences"*, see [15].

Figure 3 shows the syntax of the annotations by markers and scopes. The productions of the `C_STMT` (the C statements of ANSI C) are extended by the `SCOPE` statement, which is like the standard block statement given by curly brackets. The token `INT_CST` is used to express an integer constant.

3 Transformation of Path Information

WCET analysis for the computation of execution time bounds with high quality is in general based on the analysis of machine code. To increase readability, the machine code is usually represented as assembly source code. For the analysis of high-level programming language source code a mechanism is needed to transform the knowledge about the execution behavior of source statements into a timing description for the corresponding assembly source.

Since manual transformation is infeasible and a compiler-external transformation does not support compiler optimizations, the chosen approach integrates the transformation into the compiler. This requires several modifications of the compiler but leads to a more powerful transformation handling than the other.

The transformation method presented in this paper realizes the third approach. The compiler `GCC`[1] was enhanced with the capability of generating assembly code annotated with timing information that is derived from a WCETC source code. `GCC` was chosen, since its source code is available under GNU GPL[2].

3.1 Overview of the Compiler Architecture

The `GCC` compiler works as a single-pass compiler. It translates the source language statements to a `GCC` specific intermediate code called RTL (Register Transfer Language) [6]. RTL consists of a double linked list of imperative statements which are in the first stage hardware-independent.

Figure 4 gives a simple overview of the processing steps of `GCC` for code generation. The syntax analysis is done with assistance from the lexical analysis and generates the intermediate code for every function inside a source file. This intermediate code is then further processed in a number of steps that transform the

[1] GNU Compiler Collection

[2] General Public License, details at `http://www.gnu.org`

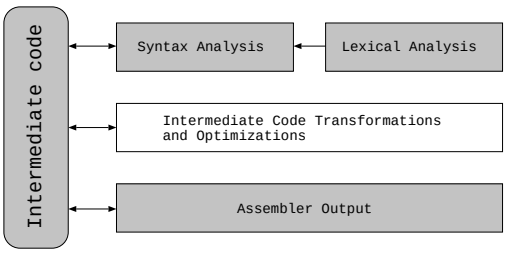

Figure 4. Compiler Architecture Overview

hardware independent representation into a specialized representation for the target hardware. Since all mappings from pseudo registers to hardware registers and stack memory are done during this transformation it is finally a straight-forward process to emit the assembly statements from the RTL representation.

3.2 Intermediate Code Statements for WCET Annotations

The information about the WCET behaviour given by the annotations in the source code is represented in note-statements in the intermediate code. These note-statements do not affect the behaviour of the compiled program and are transformed without change during the steps of compilation. If any optimizations are carried out that also effect the note-statements, the note-statements are changed, modified or deleted in parallel with the nodes that represent the code. The following note-statements are used to express the information for WCET analysis inside the intermediate code:

NOTE_INSN_WCET_MARKER(_name_): This note assigns a path name to the current edge of the program flow graph. The path names are used to express path restrictions for certain edges of the flow graph. This type of note is necessary for the WCET analysis tool.

NOTE_INSN_WCET_RESTRICTION (_restriction_): This type of note is used at the end of blocks to express restrictions to various edges of the flow graph inside the block. Here a block does not mean a basic block with strictly sequentially executed statements. A block refers to a functional block, e.g., a whole loop. There can be more than one restriction in sequential order.

NOTE_INSN_WCET_ALT_RESTRICTION (_restrictions_): This type of note is used at the end of blocks to express a set of alternative constraints to various edges of the flow graph inside the block. During WCET analysis only the restriction that leads to the worst-case execution time is selected. This requires multiple WCET evaluations of the program, depending on the number of alternative constraints for each alternative restrictions.

The following subsection describes the transformation of WCET annotations given by the programmer inside the high-level source code into the intermediate code. The type of intermediate code that is used inside the compiler to represent the WCET information is shown. The representation is given at this level, because it is hardware independent, but still similar to the representation used for WCET analysis, the assembly language representation.

The following notation is used to describe further statements that the compiler inserts into the intermediate code to describe the program path information.

code_label(_name_): This statement represents a label with a specified name.

loop-body: This covers all the statements that are representing the statements of the body of a loop.

test_code: This statement calculates the end condition of a loop. Hardware dependent details are not mentioned at this representation level.

cond_jump_label(_name_): This statement represents a conditional jump instruction to the target as given in `_name_`.

jmp_label(_name_): This statement is an unconditional jump to the given target `_name_`. A statement like this implies a *barrier* inside the sequential order of the statements. This means the predecessor of the following statement must be a jump instruction or a conditional statement. Otherwise the following statement belongs to unreachable code and can be ignored for WCET analysis.

3.3 Program Structure Transformation

This section describes the transformation of annotations in high-level constructs to a proper representation as intermediate code. The transformation is described for the `do`-loop (see Figure 5, 6 and 7). The transformation of other program structures is similar.

The iteration bounds of loops are expressed with annotations in the WCETC source code. These iteration bounds are expressed at intermediate code level by two markers and a restriction for them. Markers and

do maximum INT_CST **iterations**
C_STMT
while (EXPR) ;

Figure 5. Syntax for an Annotated do-Loop

restrictions are used to build constraints for WCET calculation which is based on integer linear programming. They are described in more depth in [18].

The names of the markers are constructed in the form LOOP_MARKER_*xx* where *xx* is a consecutive number. One marker is placed in front of the loop and the second marker is placed inside the body of the loop. The restriction on these markers is inserted after the loop. It is derived from the loop-bound annotation of the source code.

The final arrangement of the statements relating to loops depends on the optimizations performend on them. In the following example intermediate code examples are given both for full (flag -O3) and no optimizations (-O0) of the compiler.

```
NOTE_INSN_WCET_MARKER(LOOP_MARKER_x)
NOTE_INSN_LOOP_BEG
  code_label(a);
NOTE_INSN_WCET_MARKER(LOOP_MARKER_y)
  loop-body;
NOTE_INSN_LOOP_CONT
  test_code;
  cond_jump_label(b);
  jmp_label(c);
  code_label(b);
  jmp_label(a);
NOTE_INSN_LOOP_END
NOTE_INSN_WCET_RESTRICTION(LOOP_MARKER_y <= INT_CST*LOOP_MARKER_x)
  code_label(c);
```

Figure 6. Intermediate Code for an Annotated do-Loop without Optimizations

```
NOTE_INSN_WCET_MARKER(LOOP_MARKER_x)
NOTE_INSN_LOOP_BEG
  code_label(a);
NOTE_INSN_WCET_MARKER(LOOP_MARKER_y)
  loop-body;
NOTE_INSN_LOOP_CONT
  test_code;
  cond_jump_label(a);
NOTE_INSN_LOOP_END
NOTE_INSN_WCET_RESTRICTION(LOOP_MARKER_y <= INT_CST*LOOP_MARKER_x)
```

Figure 7. Intermediate Code for an Annotated do-Loop with Optimizations

Figure 5 shows the syntax of the do-loop. Figures 6 and 7 provide the corresponding intermediate code for the different optimization levels of the compiler. The comparison of these figures shows the influence of using different compiler optimization levels. The path annotations are adapted when the code structure is changed during the compilation process.

4 Experiments

```
/* processor: m68000 */
/* memory wait states (r/w):  0/ 0 */

----- CYCLES(bubble) = 47034 -----
  1|----------#define N_EL 10
  2|----------
  3|----------
  4|----------/* Sort an array of 10 elements with bubble-sort */
  5|----------void bubble (int arr[])
  6| 1|   16 -{
  7|----------  /* Definition of local variables */
  8|----------  int i, j, temp;
  9|----------
 10|----------  /* Main body */
 11| 3|   24 -  for (i=N_EL;
 12| 4|  328 -          i > 1;
 13| 2|  306 -          i--)
 14|----------          maximum (N_EL - 1) iterations
 15|----------  {
 16| 2|  180 -          for (j = 2;
 17| 4| 4032 -                  j <= i;
 18| 2| 2754 -                  j++)
 19|----------                  maximum (N_EL - 1) iterations
 20|----------          {
 21|16|13122 -                  if (arr[j-1] > arr[j])
 22|----------                  {
 23| 9| 7614 -                          temp = arr[j-1];
 24|14|11988 -                          arr[j-1] = arr[j];
 25| 6| 6642 -                          arr[j] = temp;
 26|----------                  }
 27|----------          }
 28|----------  }
 29| 2|   28 -}
```

Figure 8. Bubble-Sort Algorithm (without optimization)

```
/* processor: m68000 */
/* memory wait states (r/w):  0/ 0 */

----- CYCLES(bubble) = 8210 -----
  1|----------#define N_EL 10
  2|----------
  3|----------
  4|----------/* Sort an array of 10 elements with bubble-sort */
  5|----------void bubble (int arr[])
  6| 4|   56 -{
  7|----------  /* Definition of local variables */
  8|----------  int i, j, temp;
  9|----------
 10|----------  /* Main body */
 11|----------  scope BS
 12|----------  {
 13| 1|    4 -          for (i=N_EL;
 14| 3|  178 -                  i > 1;
 15| 1|   36 -                  i--)
 16|----------                  maximum (N_EL - 1) iterations
 17|----------          {
 18|----------                  for (j = 2;
 19|10| 1854 -                          j <= i;
 20| 2| 1296 -                          j++)
 21|----------                          maximum (N_EL - 1) iterations
 22|----------                  {
 23| 4| 3474 -                          if (arr[j-1] > arr[j])
 24|----------                          {
 25|----------                                  marker M;
 26|----------                                  temp = arr[j-1];
 27| 1|  540 -                                  arr[j-1] = arr[j];
 28| 1|  720 -                                  arr[j] = temp;
 29|----------                          }
 30|----------                  }
 31|----------          }
 32|----------          restriction M <= (N_EL*(N_EL-1)/2);
 33|----------  }
 34| 4|   52 -}
```

Figure 9. Bubble-Sort Algorithm (optimization and consideration of infeasible paths)

This section describes the WCET calculations of some sample programs. The evaluated programs are standard algorithms for sorting and searching of data in a data vector. These examples demonstrate the importance of being able to transform path annotations during compiler optimizations. Therefore the WCET analysis was performed with and without strong optimizations. The WCET compiler supports powerful annotations to describe (in)feasible paths. To demonstrate the importance of such annotations a comparison of the results between using simple iteration bounds for

Algorithm	Optimization level	
	none (00)	full (03)
bubble sort	47034	9146
bubble sort, with markers	35442	8210
selection sort, with markers	44830	14948
binary search	2014	810

Table 1. Calculated WCET [cycles] of programs

nested loops and using an advanced description of the iteration count by markers and scopes is given.

The experiments are performed by analyzing small sample programs written in WCETC (Section 2.2). Each program is analyzed twice, first without (flag `-O0`) and second with full code optimizations (flag `-O3`) performed by the compiler. The low-level WCET tool is directly called by the compiler to calculate the WCET of the compiled code. The results of the WCET tool are mapped to the corresponding lines of the source code. This mapping is called back-annotation.

For back-annotation, three data columns are added on the left side of the source file:

- The line number in the source file.
- The number of assembly instructions that have been produced for the constructs of this source line.
- The absolute number of cycles of the source line contributing to the worst-case execution time.

An additional header is placed to inform about the hardware properties of the target system. The chosen target platform was based on a MC68000 processor from Motorola.

4.1 Results

Two sorting algorithms (*bubble sort* and *selection sort*) and one search algorithm (*binary search*) were analyzed to see effects of different compiler optimizations and annotations. The *bubble sort* algorithm was analyzed both with complete and with basic annotations to show the necessity of supporting transformation of complete annotation constructs by the compiler.

The calculated WCET of all sample programs is summarized in Table 1.

It is remarkable that the calculated execution time improved up to 80 percent, when using full compiler optimizations. This stresses the necessity of taking compiler optimizations into account for WCET analysis.

The *bubble sort* algorithm was analyzed in further detail to demonstrate also the importance of supporting the transformation of the full set of annotations by the WCET analysis framework. Using only loop bounds makes WCET analysis assume that the body of the inner loop executes $(N-1)^2$ times, where N is the maximum length of the input vector. By taking into account path information that describes the asymmetric iteration count of the inner loop, WCET analysis can use the real number of executions of the inner-loop body, $N(N-1)/2$ (see Figure 9 for the corresponding source code).

Figure 8 shows the WCET results for computation without code optimizations. The further improvement of WCET using more accurate annotations by specifying additional (in)feasible paths and using compiler optimization is given in Figure 9. Note that the calculated WCET for the code with additional annotations by markers and scopes is reduced down to about 70 percent.

While it was quite straight-forward to build transformation of path information into the compiler, it would be arduous and complicated to design a WCET framework for these requirements where the structure transformation of the program is done *outside* the compiler. The optimization techniques used by a modern compiler would require an enormous set of rules to be modeled by the external transformation tool. On the other hand, the examples have shown the importance for WCET analysis to have effective annotations for modeling the runtime behaviour of a program.

5 Summary and Conclusion

We described a technique for compiling programs with path information for WCET analysis. We presented a WCET tool chain that is suitable for the integration of program path transformation into the compiler. This leads to the ability of handling more complex program code optimizations instead of just performing this transformation outside the compiler. We introduced a programming language that supports statements to annotate path information inside the program and presented a compiler concept for transforming this information down to the machine code. The programming language was derived from ANSI C with several restrictions and additional statements to support the WCET analysis.

The experiments have shown the necessity of supporting strong code optimizations during the compilation process. The importance of an effective concept

for program path annotations for the programming language was also pointed out by these experiments.

For the future work we plan to extend this WCET analysis framework to support processors with performance enhancing features such as pipelines or caches.

References

[1] R. E. Burkhard. *Methoden der ganzzahligen Optimierung*. Springer-Verlag, 1972.

[2] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2):249–274, May 2000.

[3] J. Engblom, A. Ermedahl, and P. Altenbernd. Facilitating Worst-Case Execution Time Analysis for Optimized Code. Technical report, Uppsala University, Uppsala, Sweden, 1998.

[4] H. Kopetz et al. Real-Time System Development: The Programming Model of MARS. In *Proc. of the International Symposium on Autonomous Decentralized Systems*, pages 290–299, 1993.

[5] M. Exler. Propagierung von Pfadinformation für die Analyse von Programmlaufzeiten. Master's thesis, Technische Universität Wien, Vienna, Dezember 1999.

[6] Free Software Foundation, editor. *Using and Porting GNU CC*. Number 2.7.2 in ISBN 1-882114-66-3. Free Software Foundation, Boston, USA, November 1995.

[7] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding Pipeline and Instruction Cache Performance. In *IEEE Transactions on Computers*, number 48 in 1, January 1999. old ID: HEALY:99.

[8] C. A. Healy, M. Sjödin, and D. Whalley. Bounding Loop Iterations for Timing Analysis. In *Proceedings of the IEEE Real-Time Technology and Aplications Symposium*, pages 12–21, June 1998. old ID: HEALY:IEEE98.

[9] R. Kirner. Integration of Static Runtime Analysis and Program Compilation. Master's thesis, Technische Universität Wien, Vienna, Austria, May 2000.

[10] E. Klingerman and A. Stoyenko. Real-Time Euclid: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, 12(9):941–989, September 1986.

[11] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, and W. Schütz. The Design of Real-Time Systems: From Specification to Implementation and Verification. *IEEE Software Engineering Journal*, 6(3):72–82, May 1991.

[12] Y.-T. S. Li, S. Malik, and A. Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 380–387, November 1995.

[13] D. Macos and F. Mueller. Integrating Gnat/Gcc into a Timing Analysis Environment. In *Work-in-Progress of EuroMicro Workshop on Real-Time Systems*, pages 15–18, June 1998.

[14] C. Y. Park and A. C. Shaw. Experiments with a Program Timing Tool based on a Source-Level Timing Schema. *Computer*, 24(5):48–57, May 1991.

[15] P. Puschner. Ermittlung der maximalen Abarbeitungszeit von Programmen. Master's thesis, Technische Universität Wien, Vienna, September 1988.

[16] P. Puschner. *Zeitanalyse von Echtzeitprogrammen*. PhD thesis, Technische Universität Wien, Vienna, December 1993. old ID: PUSCHNER:93.

[17] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *The Journal of Real-Time Systems*, 1:159–176, 1989.

[18] P. Puschner and A. V. Schedl. A Tool for the Computation of the Worst Case Task Execution Times. In *Proceedings Euromicro Workshop on Real-Time Systems*, pages 224–229, Oulu, Finland, June 1993.

[19] J. Reisinger. *Konzeption und Analyse eines zeitgesteuerten Betriebssystems für Echtzeitanwendungen*. PhD thesis, Technische Universität Wien, Vienna, Austria, April 1993.

[20] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[21] A. Vrchoticky. *The Basis for Static Execution Time Prediction*. PhD thesis, Technische Universität Wien, Vienna, Austria, April 1994.